INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

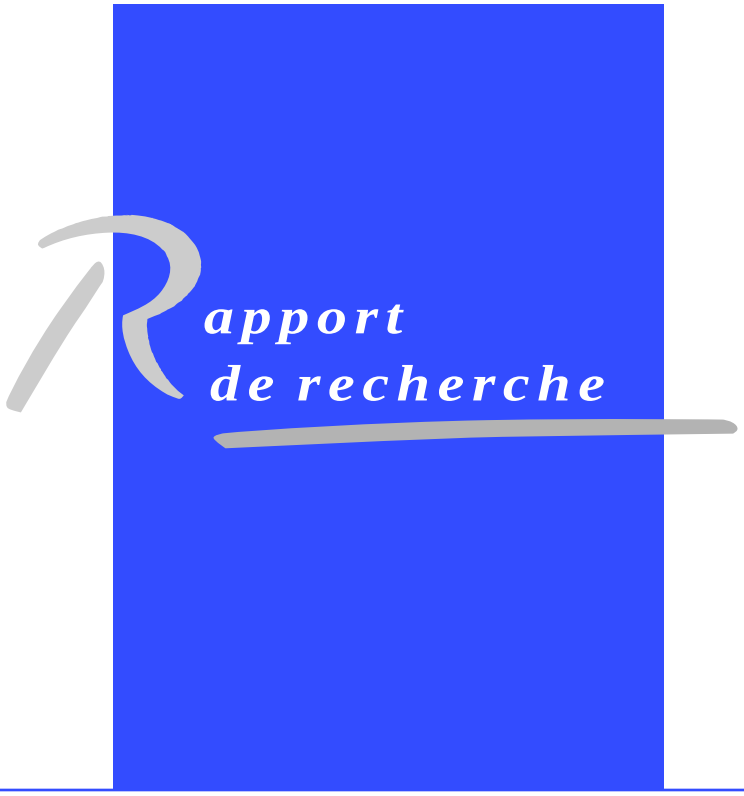
Dynamic Resource Management in a Cluster for Scalability and High-Availability

Pascal Gallard, Christine Morin, Renaud Lottiaux

N°4347

Janvier 2002

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4347--FR+ENG

Dynamic Resource Management in a Cluster for Scalability and High-Availability

Pascal Gallard, Christine Morin, Renaud Lottiaux*

Thème 1 — Réseaux et systèmes
Projet Paris

Rapport de recherche n ° 4347 — Janvier 2002 — 17 pages

Abstract: In order to execute high performance applications on a cluster, it is highly desirable to provide distributed services that globally manage physical resources distributed over the cluster nodes. However, as a distributed service may use resources located on different nodes, it becomes sensitive to changes in the cluster configuration due to node addition, reboot or failure. In this paper, we propose a generic service performing dynamic resource management in a cluster in order to provide distributed services with high availability and scalability. This service has been implemented in Gobelins cluster operating system. The dynamic resource management service we propose makes node addition and reboot nearly transparent to all distributed services of Gobelins and, as a consequence, fully transparent to applications. In the event of a node failure, applications using resources located on the failed node need to be restarted from a previously saved checkpoint but the availability of the cluster operating system is guaranteed, provided that its distributed services implement reconfiguration features.

Key-words: Cluster, distributed system, resource management, high-availability, fault-tolerance.

(Résumé : tsvp)

* {plgallar,cmorin,rlottiau}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Gestion dynamique de ressource dans un cluster pour le passage à l'échelle et la haute disponibilité

Résumé : Dans le but d'exécuter des applications à hautes performances sur une grappe de calculateurs, il est fortement souhaité de disposer d'un ensemble de services distribués capable de gérer les ressources physiques distribuées à travers les différents nœuds de la grappe. Cependant, un service distribué peut exploiter des ressources physiquement situées sur des nœuds distincts. Ce service est donc dépendant des changements de configuration de la grappe de calculateurs suite à un ajout, retrait ou défaillance d'un nœud. Dans cet article, nous proposons un service générique offrant une gestion dynamique des ressources d'une grappe dans le but d'offrir des services répartis hautement disponible et passant facilement à l'échelle. Ce service est réalisé dans le système d'exploitation pour grappe Gobelins. La gestion dynamique des ressources que nous proposons rend l'ajout ou le retrait de nœud pratiquement transparent à tous les services distribués de Gobelins et par conséquent à toutes les applications. Dans le cas de la défaillance d'un nœud, les applications exploitant des ressources présentes sur le nœud défaillant doivent être redémarrer depuis un point de reprise précédemment effectué, mais la haute-disponibilité du système est garanti, en supposant que les services distribués fournissent les outils de reconfiguration necessaires.

Mots-clé : Grappe de calculateurs, système distribué, gestion de ressource, haute-disponibilité, tolérance aux fautes

1 Introduction

To efficiently execute high performance applications, cluster operating systems must offer some global resource management services as for example a remote paging system[8], a system of cooperative file caches[11] or a global scheduler[1]. As another example, a cluster OS may offer a Distributed Shared Memory[7, 5] service to support applications based on the shared memory paradigm. A cluster OS can be defined as a set of distributed services. Due to its distributed nature, the high-availability of such an operating system is not guaranteed when a node fails. Moreover, a node addition or shutdown should be done without stopping the cluster and its running applications.

In this paper, we propose a dynamic resource management service whose main goal is to hide any kind of change (node addition, eviction or failure) in the cluster configuration to the OS distributed services and to the applications.

This work takes place in the framework of the design and implementation of Gobelins cluster OS. Gobelins is a single system image OS which aims at offering the vision of an SMP machine to programmers. Gobelins implements a set of distributed services for the global management of memory, processor and disk resources. Our generic dynamic resource management service has been experimented with the global memory management service (a Distributed Shared Memory)[10] of Gobelins for node addition and eviction.

The remainder of this paper is organized as follows. In the next section we introduce the model of the distributed services considered in this paper and present our assumptions. In Section 3 we describe the proposed dynamic resource management service and show how it fits in Gobelins OS architecture. Section 4 provides some details related to the service implementation and presents experimental results. Section 5 presents related works and we conclude in Section 6.

2 Model of System and Assumptions

We consider in this paper a cluster OS as a set of distributed services. Each service provides global management for one kind of physical resource. For example, a cluster OS may offer a DSM service, providing a form of global memory management[9] or a Parallel File System [4] which provides global disk management.

2.1 Model of a Distributed Service

In this section, we propose a general model of a distributed service, part of a cluster operating system. Gobelins OS distributed services conform to this model. We take the example of a distributed shared memory to illustrate this model.

The cluster is composed of a set of N nodes numbered from 0 to $N-1$. Let us consider a service S, in charge of the management of a set of physical resources of type R to implement an operating system abstraction A. For example, a DSM system manages a set of page frames in physical memory to implement the virtual page abstraction.

Each node i of the cluster may have local instances of the physical resource. Instances of resource R on node i are managed by a local server L_i which is also in charge of processing requests sent by remote nodes and related to the local resources. In the DSM example, a page server is a local server. It responds to page invalidation or page requests.

In a distributed system, global information is associated with each instance of an abstraction. This information may contain the identity of nodes that are concerned by the implementation of the abstraction on top of the physical resources. Global information is kept in a directory indexed with the abstraction instance number. Directory information changes over time as responsibility of nodes regarding the management of a particular abstraction changes. There are several ways to manage the directory of a given service:

1. In a centralized approach, a node i is in charge of maintaining the whole directory; a drawback of this approach is that node i may become a bottleneck as it is contacted each time a node needs to access global information.

2. In an approach based on redundancy, the directory is replicated on all nodes. The drawback of this approach is that updating global information is very expensive as all copies of the directory need to be kept consistent.

3. In a distributed approach, each node is in charge of maintaining a subset of the directory.

The node in charge of a particular directory entry is generally statically defined. This approach does not suffer from the drawbacks of the two previous approaches. In our model, we consider a distributed approach. Each node hosts a manager which is a process in charge of managing directory information.

For example, in a DSM system, the owner of a page is the node that has a copy of the page in a local page frame and is the last one to have written the page. A directory maintains for each virtual page the identity of its current owner as the owner of a given page may change over time. Manager of virtual page j can be defined statically as being node $j \ mod \ N$.

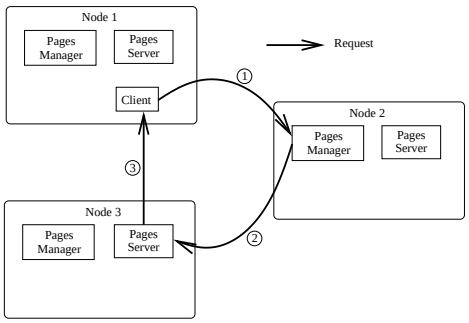


Figure 1: *Example of a distributed service*

In a cluster formed by three nodes (see Figure 1), let us assume that a client process, running on Node 1, makes a *read request* on a missing page p. In order to obtain a copy of p, Node 1 must locate the manager of p and send it the *read request*. Node 2 which is p manager forwards the request to Node 3 that stores the requested page. The local server of Node 3 completes the request and sends the page to Node 1.

2.2 Assumptions

We call configuration the set of active nodes in the cluster. A node is considered to be active if it has not been detected failed by other active nodes and is not currently being added to or evicted from the cluster. The system is said to be in *stable state* when no configuration change is being processed. For the sake of clarity, we assume that only one modification in the cluster configuration (a node addition, eviction or failure) may occur at any time. In this context, we assume that a configuration change only happens when the system is in *stable state*. Thus a node failure cannot happen when the cluster OS is processing a node addition or eviction. We assume that the network is never partitioned, as a network partition would lead to the failure of several nodes at the same time.

We assume that the communication system guarantees that there is no message lost or duplicated messages and that messages are delivered in the order they are sent. Moreover, we assume that messages are not altered during transmission.

2.3 Impact of a Cluster Configuration Change on a Distributed Service

Let us now describe what the impact of a node addition, eviction or failure is on a distributed service.

2.3.1 Node addition

All the cluster nodes must be informed that a new node is added in order to be able to communicate with it. On the other side, the new node must learn the current configuration of cluster. A new node provides additional physical resources in the cluster. Therefore, a new local server must be created on the added node. This node may also participate to the global management of the service. In this case a new manager needs to be created. Managers executing on other nodes must be informed of the addition of a new node in order to send it part of the directory. When the node addition processing is completed, all active nodes of the cluster must have the same view of the cluster configuration.

2.3.2 Node eviction

When a node is shut down, there is less physical resources in the cluster. The corresponding local server is stopped. In this context, the cluster OS must stop the usage of these resources and nodes must stop to send messages to the evicted local server. Moreover, part of the

directory may be stored in the node shutting down. So these directory entries must be migrated onto one or several other nodes. In order to balance the load of managing directory entries on all managers, it may be useful to compute a new directory distribution function. The clients requests need to be transmitted to the right manager despite the migration of some directory entries. As the set of physical resources implementing a virtual resource may change when a node eviction happens, some directory entries must be updated to reflect the new way a virtual resource is implemented. Finally, if processes are executing on the stopping node, they must be migrated onto other active nodes. As for a node addition, the node leaving the cluster must inform other nodes in order that all remaining active nodes have the same view of the new cluster configuration.

2.3.3 Node Failure

A main difference between a node failure and the two previous situations is that a failing node cannot notify other nodes of its failure. So it is necessary that all active cluster nodes detect a node failure before any recovery action. When a node failure occurs, the cluster loses the local resources of the failed node, a manager and the directory entries it was managing. The directory information on some nodes may not be valid anymore, and has to be reconstructed during recovery. Processes executing on the failed node are lost and should be restarted from a checkpoint after recovery, if any exists. Moreover, some messages sent to the failed node cannot reach their destination.

3 Dynamic Resource Management

In this section, we present a generic dynamic resource management service that makes changes in the cluster configuration nearly transparent to the OS distributed services.

Node addition and shutdown are made completely transparent to the applications assuming that a process migration mechanism is provided. A node failure is also transparent for checkpointed applications.

3.1 Design of a Dynamic Resource Management Service

Our goal is to design a layer in the cluster OS, that we call adaptation layer, which is in charge of managing configuration changes in the cluster and to trigger reconfiguration of distributed services when needed (for example, after detection of a node failure). In the proposed architecture, the OS distributed services rely on the adaptation layer which is based on the cluster communication system (Figure 2). The adaptation layer is designed to make reconfiguration changes transparent to any kind of distributed service.

The way distributed services are programmed is unchanged. A generic interface has been defined between distributed services and the adaptation layer. In addition we want to have a generic protocol to handle any kind of configuration change in a cluster. We do not want to design as many different mechanisms as particular changes in the cluster configuration.

The adaptation layer is in charge of the following tasks:

1. implementing a protocol for a node to inform other nodes that it is added or shut down,
2. detecting node failures,
3. providing a consistent view of the cluster configuration,
4. coordinating the protocol to deal with a cluster configuration change,
5. computing a distribution function that keeps the directory entries management balanced between managers,
6. locating the manager of a particular entry of the directory (with a *locator*),
7. dealing with migration of directory entries when a configuration change happens.

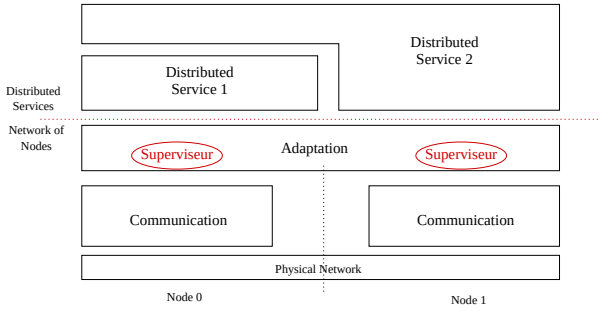


Figure 2: *Dynamic architecture model*

The adaptation layer must be as independent as possible from a specific service. However, directory entries management is obviously performed by the related distributed service. We need to have some generic operations like acknowledgement, and specific operations for each service. Moreover, when a change occurs in the configuration all services need to perform some reconfiguration. In this way, for each service similar operations is needed at the same time.

A distributed service can use the adaptation layer after a registering step. The specific functions are declared during this registering.

This is the main reason why we split the adaptation layer into two parts. First, a global mechanism defines some generic operations inside the adaptation layer. Specific service tasks is provided by a plug-ins system.

One important task to be performed when a configuration change occurred is the migration of directory entries. In the adaptation layer the directory entries migration is performed in four parts (Figure 3) during an *adaptation period*. Each step may call a specific service function (provided by the plug-ins system).

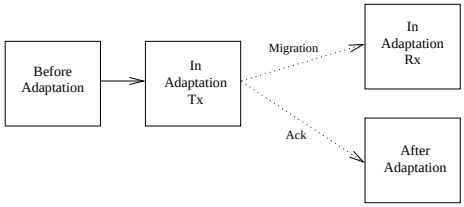


Figure 3: *Adaptation model*

3.1.1 Before migration

The adaptation layer prepares the migration of directory information. This stage happens before updating the locator.

3.1.2 Sending out

The locator is assumed updated. For each services, the directory entries that must change manager are migrated to the corresponding nodes. A general acknowledgement sent to all nodes notices the end of this emission step.

3.1.3 Receiver

This step is the counterpart of the previous one. Directory entries are received by nodes and prepared to be merged with the set (that may be empty on an added node) of local directory entries of the receiving node. Acknowledgements are checked.

3.1.4 After migration

When all the acknowledgements are received, each node builds its local part of the directory by merging its remaining entries and the received entries. At the end of this step, the directory entries migration stage of this node is ended.

3.2 Interface: the Example of the Global Memory Management System

For the sake of clarity, we choose to describe the specific functions that a service has to provide on the example of a DSM service.

In the considered service, a directory entry corresponds to a virtual page. Let us take the sample of a cluster with three nodes. We assume that a copy of Page 42 is located on Node 3 in a page frame, and its directory information is managed by Node 2.

Let us consider a page fault (read request) on Page 42 on Node 1. Node 1 (as client) calls: `adaptation_send(PageReadRq, 42)`. In this context, the client makes a request on a resource and it does not try to know where the request goes.

On another side, the specific registered functions allow to handle the page information and perform the necessary work as described above.

3.3 Adaptation Layer

At any time (*adaptation period* or *stable state*) several parts of the adaptation layer take parts in the proper operation of the cluster. These different parts are described in the remainder of this section.

3.3.1 Supervisor

A supervisor process is executed on each node. It is responsible of the addition or the shutdown of the node on which it executes. This is the supervisor that prepares its own node and notices the cluster.

The set of supervisors in the cluster cooperate in order to maintain a consistent view of the cluster configuration. In this way, a node *supervisor* participates to the failure detection protocol. When a node failure happens (or is suspected) a consensus protocol, which is out of the scope of this paper, is executed.

The supervisor (see Figure 4) starts with the setting and the addition of its node in the cluster (stage 1, 2, 3). The *stable state* (4) corresponds to a node outside an addition, eviction or failure stage. In a second time, the supervisor takes part of the addition of a new node (stage 5) and the eviction or failure detection (stage 6). The supervisor is the link between the node and the global state of the cluster, defined by the consensus protocol.

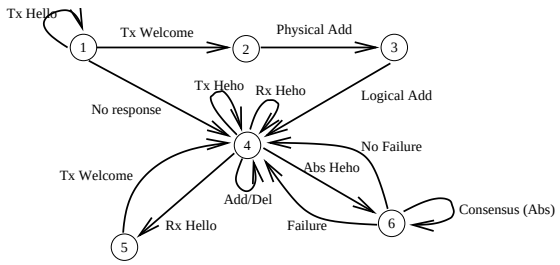


Figure 4: *Supervisor state*

When a configuration change happens in the cluster (Figure 5), the supervisor makes an update of its communication layer (state 2) and, if needed, on the physical communication system. The next step (state 3) is the logical update, where the supervisor triggers directory entries migration. After migration, the supervisor executes a consensus protocol and retrieves a *stable state*.

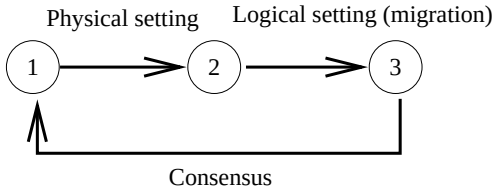


Figure 5: *Cluster state*

3.3.2 Locator

The aim of the locator is to keep track of the node that is the manager of each directory entry. It must be quick to retrieve a directory entry.

A first approach is to assign directory entry managers to nodes with a *distribution function* like *modulo*. Locating a particular directory entry manager is very simple and fast. However when a configuration change happens, the migration stage may affect all the nodes in the cluster and directory entries may be exchanged between two nodes that were already active before the configuration change[6]. When the cluster has got a large number of nodes, the directory entries migration step may become inefficient. That approach is not scalable.

In order to address scalability issue, we choose to split up the whole set of nodes, in several subsets with a limited size called cells. A tree of cells organizes the cells themselves. Each cell may contain other cells or nodes. The aim of a cell is to be a load-balancing unit: a bunch of directory entries is affected to a cell and when the composition of a cell is changed, the bunch of entries is re-distributed among the members of the cell. In this way, the distribution of directory entries affects only a few nodes in the cluster.

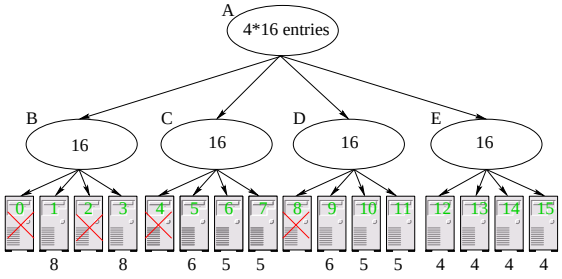


Figure 6: *Tree cell sample*

In Figure 6, we have the example of a 2 level tree which corresponds to a cluster of 12 nodes. Each cell manages 16 directory entries. By assumption, Cell A manages 64 entries on 4 cells. There are 3 or 4 nodes affected to each of these cells.

The same tree is used to distribute directory entries of all services existing in the cluster. In order to maximize the use of the different cells, the size of a service can be declared at its

initialization and a particular service may be affected to a sub-tree instead of being affected to the whole tree. With this mechanism several small (in term of number of directory entries) services can be deployed on different nodes. Finally, the use of an array to store the deployment in a cell, makes the localization of the manager in a cell of a directory entry, constant in time.

The most interesting feature is that when a configuration change happens in the cluster, only one cell is affected by the operation. That way, the number of directory entries that migrate is limited. Moreover, in case of an addition, the most appropriate cell can be chosen in order to have the best load-balancing policy.

3.3.3 Manager

As explained previously, there is a manager on each node and it acts like a directory for a subset of the resources. When a resource instance changes or is migrated, the associated directory entry must be updated by the corresponding manager in the appropriate distributed service.

4 Implementation in Gobelins and Evaluation

The dynamic resource management service described in the previous sections has been implemented in Gobelins cluster OS and has been experimented with Gobelins global memory management service which is an example of distributed service.

4.1 Implementation

In the remainder of this section, we present the adaptation layer functions for the different cases of cluster configuration changes. We distinguish two kinds of events: foreseeable ones, namely node addition or eviction, and node failures. Protocols implemented in the adaptation layer are described from two points of view: that of the node added to or evicted from the cluster, which we call requesting node, and that of any other active node in the cluster, which we call an observing node.

4.1.1 Foreseeable events

In the event of a node addition or eviction, directory entries need to be migrated, in order to balance the number of directory entries managed by each node. When a configuration change is announced in the cluster, several communications may occur between nodes. The destinations of some of these communications may have changed due to the eviction of the destination node or the new directory entries distribution. These ongoing communications are delayed and not aborted. In this way, running applications do not notice the configuration change.

Requesting node In the foreseeable case, the node notifies every active node in the cluster about its situation. In all cases, it is able to prepare itself and starts the directory entries migration process. This migration is performed respectively during the initialization or finalization step of the requesting node. An added node only receives data from other node, while an evicted node only transmits data to other nodes.

Observing node The situation for an observing node is quite similar. When a notification of configuration is received, the directory entries migration phase is started. This migration happens in the *stable state* of the node.

Migration step Migration details are represented in Figure 7. In the *stable state*, a node may receive (states 1, 2, 3) some directory entries from other nodes. This is the case of a node receiving directory entries while it has not yet received notification of the beginning of the configuration step. When a node receives the configuration change notification (state 4), it may receive other directory entries but also starts to transmit its own directory entries (states 5, 6). At that time the adaptation layer makes transmissions, in a sequential way, for all registered services with the provided functions. Destination is one or several other nodes depending on the new directory entries distribution. At the end of the sending out step, the adaptation layer sends to all nodes (including itself) an acknowledgement. The next step is to wait for directory entries (step 7) until all the *Ack* have been received. After receiving all acknowledgements, a node can finish the integration of the new data and recover its *stable state*.

From the point of view of the cluster, the configuration change ends when all the nodes finish their own part of migration protocol.

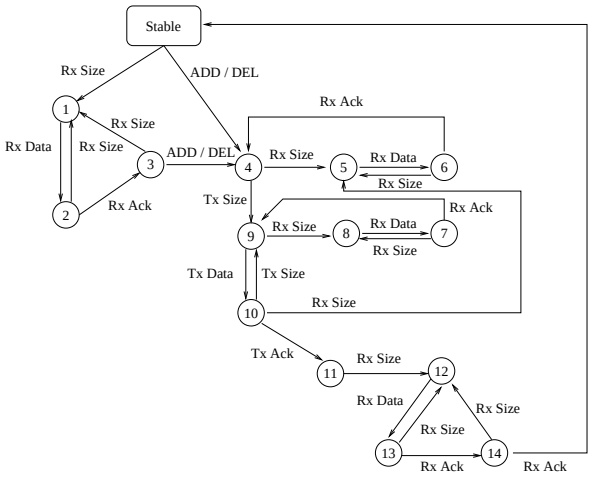


Figure 7: *Node modification state*

4.1.2 Node failure

Similarly to an eviction, a node failure may happen while several communications are occuring between active nodes and the failed node. In order for the failure to remain hidden from the applications, the communication may be interrupted and the corresponding request must be transmitted to the directory entry manager host.

In the cluster, each node stores some data (for instance page copies) and directory information. In a first step, we only focus on the recovery of directory information, recovering data is to be handled by each particular service.

There are mainly two ways to recover directory information: using redundant directory information or rebuilding directory information at recovery. Managing redundant directory entries implies overheads during normal functioning. With the rebuilding method an overhead is incurred only when there is a change in the cluster configuration. In fact, changes in the cluster configuration are generally far less frequent than updates of directory information.

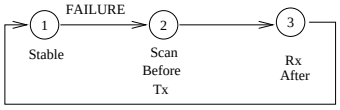


Figure 8: *Node failure state*

When all supervisors of active nodes agree on a failure, each node must scan its local directory information. This operation is performed by a "scan" function registered by each distributed service and automatically called by the adaptation layer. When orphan data (for example a memory page previously managed by the failed node) is found on a node, this node temporarily takes the management of this data and rebuilds the corresponding directory entry. The next operations (see Figure 8) are the same as for a foreseeable change: the tree is updated and migration of directory information then takes place.

4.2 Evaluation of the dynamic resource management service in Gobelins

The cluster used for experimentation is made up of four *Pentium III* (500MHz, 512KB L2 cache) nodes with 512MB of memory. The nodes communicate with a Gigabit network. The Gobelins system used is an enhanced 2.2.13 Linux kernel. We consider here two of the Gobelins modules, the high performance communication system, and the global memory management system. The adaptation layer is experimented in a modified version of Gobelins that includes an additional module implementing the adaptation layer.

We used the Modified Gram-Schmidt (MGS) algorithm as a test parallel application. The MGS algorithm produces from a set of vectors an orthonormal basis of the space generated by these vectors. The algorithm consists of an external loop running through columns producing a normalized vector and an inner loop performing for each normalized vector a

scalar product with all remaining ones. Time is measured on the external loop of the MGS program. Each test is repeated 10 times. We suppress the different error checking in the communication layer.

4.2.1 Execution measurement

In the first version of Gobelins, the directory entry managers were located by a static function based on *modulo*. During a normal execution, this function is called many times, for example to locate a page memory manager. So the efficiency of this *locate* function impacts on the global efficiency of Gobelins operating system. We made several sets of experiments with different matrix sizes (64, 128, 256, 512, 1024 and 2048) on different clusters (2, 3 and 4 nodes). For comparison, two versions of Gobelins were used: one based on *modulo* (STAT), one based on the adaptation layer (DYN). The overhead (in Figure 9) is calculated between these two measurements: $overhead = (\frac{DYN}{STAT} - 1) * 100$.

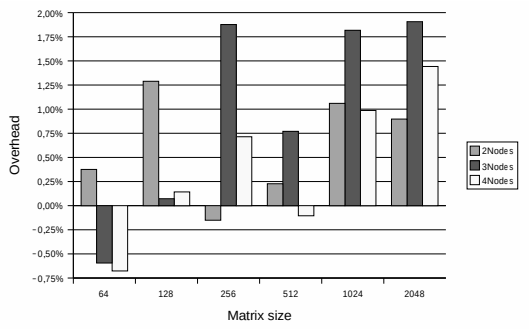


Figure 9: *Static / Dynamic overhead*

In all cases, the overhead is less than 2%. In four cases (64-3N, 64-4N, 256-2N and 512-4N), the *dynamic version* is more efficient than the *static version*. As we indicate previously, the *static version* uses a distribution based on *modulo*. On another side, the *dynamic version* uses its own distribution that is different from *modulo*. In the particular case of Gram-Schmidt, the new distribution decreases the number of page requests across the network. Cluster with two nodes and four nodes are similar cases because in these configurations every node has exactly the same number of entries to manage. The worst case is a cluster with three nodes. In this case, the load of a virtual fourth node is spread on the three nodes. This repartition leads to a non-uniform load between the nodes.

Another important factor when we evaluate the adaptation layer is the time of a node addition. A node arriving in a running cluster has to perform mainly three steps:

- physical addition in the cluster (PHYS),
- node configuration (SETTING),

- logical addition in the cluster (LOG).

Start init (START) and *end init* (END) represent two additional minor steps before and after these three main steps. The measurement protocol is the following. We start a cluster with two nodes and we run Gram-Schmidt program (1024×1024). During the execution, we first add a third node then a fourth node. For each step and for each node, we keep the best time measured. Results of the measurement are exposed in the Figure 10. In all cases, all steps but *logical addition* stay nearly constant. *Logical addition* justifies the differences. When there is no running application, no memory page manager needs to change. However, when the addition occurs during a running application, the new node advert the others in sequentially and managers control data that need to be exchanged. This is the reason why the addition of the fourth node is more expansive. This case represents nearly a third of a normal execution, but configuration changes in the cluster could be assumed to be rare. Moreover, the cluster partition made by the cells tree, limits the disruptive factor in the cluster to only the applications that depend on nodes belonging to the changing cell.

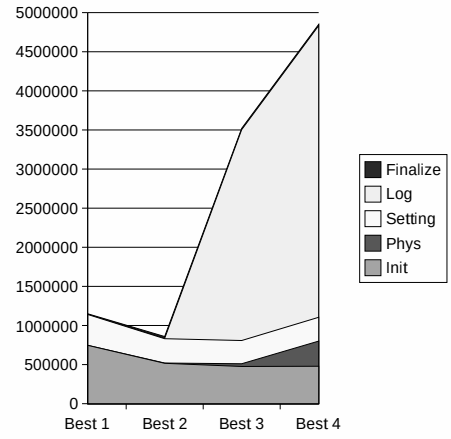


Figure 10: *Time measurement of a node addition*

The execution time of the running application has been measured and is 20.6364s (at the best effort). The cluster formed resulting from this operation distributes the processes on two nodes and the memory on three nodes. These measures are to be compared with the previously measured ones in the case of two node or three node clusters. Such a cluster has worst results than a three node cluster because there is only two "computing" nodes. Moreover, measure is worst than two nodes cluster because the distributed among three nodes of the memory managers. In these two cases, the latency to contact the manager is increased.

5 Related Work

Based on our knowledge, there is no global approach of high-availability and fault-tolerance system in a dynamic cluster operating system. However, several works exist on particular system.

The Ambers system[2] provides a programming model that allows the migration of some programming objects in a network of multiprocessors. In this way, a dynamic model has been developed[3] in order to address the foreseeable configuration changes in the network. This model provided the location of remote object with a *forwarding addresses* mechanism. However, only the foreseeable configuration change is allowed by this mechanism.

xFS is a file system design to be distributed among several physical disks. xFS acts like a directory manager to the files and all the inode stored in the filesystem. Each node in the cluster manage a part of the directory entries. The usage of a RAID-like system allows xFS to be fault-tolerant and a data stored on xFS can be retrieve after a node failure. Moreover, each manager store his management data in the filesystem itself. In this way, data can be highly available in the cluster.

6 Conclusion

We have proposed an architecture for a cluster OS allowing to implement both global and dynamic resource management. The problem of dynamic changes in a cluster configuration is addressed by the adaptation layer. This is a generic layer that makes configuration changes transparent to distributed services performing global resource management. This layer is implemented in the cluster operating system, so any service in the kernel (or in user-land) can take advantages of its features: global management memory system, global management process, global synchronization (barrier, distributed lock), distributed file system...

The proposed approach has been implemented in Gobelins OS. An existing global memory management distributed service has been used to validate the adaptation layer. This existing service was designed and implemented without taking into account dynamic resource management. Only few programming modification were needed in order to allow this service to tolerate a change in the cluster configuration.

The tree managed in the locator can be enhanced in order to offer scalability and to allow very large clusters or even cluster of clusters. In this way, node in the same physical area could be in the same cell, while nodes linked by a slow network could be placed in different cell in the tree. Otherwise, the independence of the cells in a tree let simultaneous addition or eviction nodes in the cluster at the same time, under the condition that one cell is not concerned by more than a change at a time.

Another foreseeable evolution of the adaptation layer consists in a closer integration with a checkpoint/restart mechanism in order to offer a fully fault tolerant system allowing not only the operating system but also applications to survive failures.

References

[1] A. Barak and 0. La'adan. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[2] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.

[3] M. J. Feeley, B. N. Bershad, J. S. Chase, and H. M. Levy. Dynamic node reconfiguration in a parallel-distributed environment. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 114–121, Williamsburg, VA, April 1991.

[4] J. V. Huber, C. L. Elford, D. A. Reed, and A. A. Chien. PPFS: A high performance portable parallel file system. In *Conference proceedings of the 1995 International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[5] P. Keleher. Threadmarks: Distributed shared memory on standard workstations, 1994.

[6] A. Kermarrec, C. Morin, and M. Banatre. implementation and evaluation of icare: an efficient recoverable dsm, 1998.

[7] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[8] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[9] C. Morin and R. Lottiaux. Global resource management for high availability and performance in a DSM-based cluster. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[10] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[11] Thomas E. Anderson, Michael D. Dhalin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM*, 14(1):41–79, February 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399